(2)

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

Public Reporting Burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

AD-A238 079

1. REPORT DATE

3. REPORT TYPE AND DATES COVERED
Final: 09 Jan 1991 to 01 Mar 1993

4. TITLE AND SUBTITLE
Tartan Inc., Tartan Ada VMS/C30 version 4.0, VAXstation 3100 (Host) to TI TMS320C30 VMS 5.2 (Target), 901210I1.11121

5. FUNDING NUMBERS

6. AUTHOR(S)
IABG-AVF
Ottobrunn, Federal Republic of Germany

DTIC
ELECTE
JUL 02 1991
S D
C

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
IABG-AVF, Industrieanlagen-Betriebsgeselschaft
Dept. SZT/ Einsteinstrasse 20
D-8012 Ottobrunn
FEDERAL REPUBLIC OF GERMANY

8. PERFORMING ORGANIZATION REPORT NUMBER
IABG-VSR 080

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
United States Department of Defense
Pentagon, Rm 3E114
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
Tartan Inc., TArtan Ada VMS/C30, Version 4.0, Ottobrunn Germany, VAXstation 3100 (Host) to TI TMS320C30 VMS 5.2 (Target), ACVC 1.11.

91-03866

14. SUBJECT TERMS
Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFED	UNCLASSIFIED	

NSN 7540-01-280-550

Standard Form 298, (Rev. 2-89)
Prescribed by ANSI Std. 239-128

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on December 10, 1990.

Compiler Name and Version: Tartan Ada VMS/C30 version 4.0

Host Computer System: VAXstation 3100 VMS 5.2

Target Computer System: 320C30 on Texas Instruments Application board

See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 901210I1.11121 is awarded to Tartan Inc. This certificate expires on 1 March, 1993.

This report has been reviewed and is approved.

Michael Tonndorf

IABG, Abt. ITE
Michael Tonndorf
Einsteinstr. 20
W-8012 Ottobrunn
Germany

for

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

John P. Solomond

Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

QUALITY INSPECTED 4

AVF Control Number: IABG-VSR 080
9 January, 1991

== based on TEMPLATE Version 90-08-15 ==

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 901210I1.11121
Tartan Inc.
Tartan Ada VMS/C30 version 4.0
VAXstation 3100 => TI TMS320C30
VMS 5.2 Application board

Prepared By:
IABG, ABT. ITE

DECLARATION OF CONFORMANCE

Customer: Tartan, Inc.

Certificate Awardee: Tartan, Inc.

Ada Validation Facility: IABG

ACVC Version: 1.11

Ada Implementation:

Ada Compiler Name and Version: Tartan Ada VMS/C30 Version 4.0

Host Compiler System: VAXstation 3100 VMS 5.2

Target Computer System: 320C30 on TI Application Board

Declaration:

[I/we] the undersigned, declare that [I/we] have no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A ISO 8652-1987 in the implementation listed above.

Customer Signature

Date: 12/14/90

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in Section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see Section 2.1) and, possibly some inapplicable tests (see Section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.
Conformity	Fulfillment by a product, process or service of all requirements specified.

Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is November 21, 1990.

E28005C	B28006C	C34006D	C35702A	B41308B	C43004A
C45114A	C45346A	C45612B	C45651A	C46022A	B49008A
A74006A	C74308A	B83022B	B83022H	B83025B	B83025D
B83026B	B85001L	C83026A	C83041A	C97116A	C98003B
BA2011A	CB7001A	CB7001B	CB7004A	CC1223A	BC1226A
CC1226B	BC3009B	AD1B08A	BD1B02B	BD1B06A	BD2A02A
CD2A21E	CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A
CD2B15C	BD3006A	BD4008A	CD4022A	CD4022D	CD4024B
CD4024C	CD4024D	CD4031A	CD4051D	CD5111A	CD7004C
ED7005D	CD7005E	AD7006A	CD7006E	AD7201A	AD7201E
CD7204B	BD8002A	BD8004C	CD9005A	CD9005B	CDA201E
CE2107I	CE2117A	CE2117B	CE2119B	CE2205B	CE2405A
CE3111C	CE3116A	CE3118A	CE3411B	CE3412B	CE3607B
CE3607C	CE3607D	CE3812A	CE3814A	CE3902B	

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

The following 285 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113F..Y (20 tests)	C35705F..Y (20 tests)
C35706F..Y (20 tests)	C35707F..Y (20 tests)
C35708F..Y (20 tests)	C35802F..Z (21 tests)
C45241F..Y (20 tests)	C45321F..Y (20 tests)
C45421F..Y (20 tests)	C45521F..Z (21 tests)
C45524F..Z (21 tests)	C45621F..Z (21 tests)
C45641F..Y (20 tests)	C46012F..Z (21 tests)

The following 21 tests check for the predefined type SHORT_INTEGER:

C35404B	B36105C	C45231B	C45304B	C45411B
C45412B	C45502B	C45503B	C45504B	C45504E
C45611B	C45613B	C45614B	C45631B	C45632B
B52004E	C55B07B	B55B09D	B86001V	C86006D
CD7101E				

The following 21 tests check for the predefined type LONG_INTEGER:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45612C	C45613C	C45614C	C45631C	C45632C
B52004D	C55B07A	B55B09C	B86001W	C86006C
CD7101F				

C35404D, C45231D, B86001X, C86006E, and CD7101G check for a predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER.

C35713B, C45423B, B86001T, and C86006H check for the predefined type SHORT_FLOAT.

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

A35801E checks that FLOAT'FIRST..FLOAT'LAST may be used as a range constraint in a floating-point type declaration; for this implementation, that range exceeds the range of safe numbers of the largest predefined floating-point type and must be rejected. (see 2.3.)

C45531M..P (4 tests) and C45532M..P (4 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater; for this implementation, there is no such type.

C45536A, C46013B, C46031B, C46033B, and C46034B contain 'SMALL representation clauses which are not powers of two or ten.

C45624A and C45624B are not applicable as MACHINE_OVERFLOWS is TRUE for floating-point types.

B86001Y checks for a predefined fixed-point type other than DURATION.

CA2009A, CA2009C..D (2 tests), CA2009F and BC3009C instantiate generic units before their bodies are compiled; this implementation creates a dependence on generic units as allowed by AI-00408 & AI-00506 such that the compilation of the generic unit bodies makes the instantiating units obsolete. (see 2.3.)

CD1009C uses a representation clause specifying a non-default size for a floating-point type.

CD2A53A checks operations of a fixed-point type for which a length clause specifies a power-of-ten type'small; this implementation does not support decimal smalls. (see 2.3.)

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use representation clauses specifying non-default sizes for access types.

CD2B15B checks that STORAGE_ERROR is raised when the storage size specified for a collection is too small to hold a single value of the designated type; this implementation allocates more space than what the length clause specified, as allowed by AI-00558.

The following 264 tests check for sequential, text, and direct access files:

CE2102A..C (3)	CE2102G..H (2)	CE2102K	CE2102N..Y (12)
CE2103C..D (2)	CE2104A..D (4)	CE2105A..B (2)	CE2106A..B (2)
CE2107A..H (8)	CE2107L	CE2108A..H (8)	CE2109A..C (3)
CE2110A..D (4)	CE2111A..I (9)	CE2115A..B (2)	CE2120A..B (2)
CE2201A..C (3)	EE2201D..E (2)	CE2201F..N (9)	CE2203A
CE2204A..D (4)	CE2205A	CE2206A	CE2208B
CE2401A..C (3)	EE2401D	CE2401E..F (2)	EE2401G
CE2401H..L (5)	CE2403A	CE2404A..B (2)	CE2405B
CE2406A	CE2407A..B (2)	CE2408A..B (2)	CE2409A..B (2)
CE2410A..B (2)	CE2411A	CE3102A..C (3)	CE3102F..H (3)
CE3102J..K (2)	CE3103A	CE3104A..C (3)	CE3106A..B (2)
CE3107B	CE3108A..B (2)	CE3109A	CE3110A
CE3111A..B (2)	CE3111D..E (2)	CE3112A..D (4)	CE3114A..B (2)
CE3115A	CE3119A	EE3203A	EE3204A
CE3207A	CE3208A	CE3301A	EE3301B
CE3302A	CE3304A	CE3305A	CE3401A
CE3402A	EE3402B	CE3402C..D (2)	CE3403A..C (3)
CE3403E..F (2)	CE3404B..D (3)	CE3405A	EE3405B
CE3405C..D (2)	CE3406A..D (4)	CE3407A..C (3)	CE3408A..C (3)
CE3409A	CE3409C..E (3)	EE3409F	CE3410A
CE3410C..E (3)	EE3410F	CE3411A	CE3411C
CE3412A	EE3412C	CE3413A..C (3)	CE3414A
CE3602A..D (4)	CE3603A	CE3604A..B (2)	CE3605A..E (5)
CE3606A..B (2)	CE3704A..F (6)	CE3704M..O (3)	CE3705A..E (5)
CE3706D	CE3706F..G (2)	CE3804A..P (16)	CE3805A..B (2)

CE3806A..B (2) CE3806D..E (2) CE3806G..H (2) CE3904A..B (2)
CE3905A..C (3) CE3905L CE3906A..C (3) CE3906E..F (2)

CE2103A, CE2103B and CE3107A require NAME_ERROR to be raised when an attempt is made to create a file with an illegal name; this implementation does not support external files and so raises USE_ERROR. (see 2.3.)

2.3 TEST MODIFICATIONS

Modifications (see Section 1.3) were required for 106 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B22003A	B24007A	B24009A	B25002B	B32201A	B33204A
B33205A	B35701A	B36171A	B36201A	B37101A	B37102A
B37201A	B37202A	B37203A	B37302A	B38003A	B38003B
B38008A	B38008B	B38009A	B38009B	B38103A	B38103B
B38103C	B38103D	B38103E	B43202C	B44002A	B48002A
B48002B	B48002D	B48002E	B48002G	B48003E	B49003A
B49005A	B49006A	B49006B	B49007A	B49007B	B49009A
B4A010C	B54A20A	B54A25A	B58002A	B58002B	B59001A
B59001C	B59001I	B62006C	B67001A	B67001B	B67001C
B67001D	B74103E	B74104A	B74307B	B83E01A	B83E01B
B85007C	B85008G	B85008H	B91004A	B91005A	B95003A
B95007B	B95031A	B95074E	BC1002A	BC1109A	BC1109C
BC1206A	BC2001E	BC3005B	BD2A06A	BD2B03A	BD2D03A
BD4003A	BD4006A	BD8003A			

E28002B was graded inapplicable by Evaluation and Test Modification as directed by the AVO. This test checks that pragmas may have unresolvable arguments, and it includes a check that pragma LIST has the required effect; but for this implementation, pragma LIST has no effect if the compilation results in errors or warnings, which is the case when the test is processed without modification. This test was also processed with the pragmas at lines 46, 58, 70 and 71 commented out so that pragma LIST had effect.

A35801E was graded inapplicable by Evaluation Modification as directed by the AVO; the compiler rejects the use of the range FLOAT'FIRST..FLOAT'LASTas the range constraint of a floating-point type declaration because the bounds lie outside of the range of safe numbers (cf. LRM 3.5.7(12)).

Tests C45524A..E (5 tests) were graded passed by Test Modification as directed by the AVO. These tests expect that a repeated division will result in zero; but the standard only requires that the result lie in the smallest safe interval. Thus, the tests were modified to check that the result was

within the smallest safe interval by adding the following code after line 141; the modified tests were passed:

```
ELSIF VAL <= F'SAFE_SMALL THEN COMMENT ("UNDERFLOW SEEMS GRADUAL");
```

C83030C and C86007A were graded passed by Test Modification as directed by the AVO. These tests were modified by inserting "PRAGMA ELABORATE (REPORT);" before the package declarations at lines 13 and 11, respectively. Without the pragma, the packages may be elaborated prior to package report's body, and thus the packages' calls to function Report.Ident_Int at lines 14 and 13, respectively, will raise PROGRAM_ERROR.

B83E01B was graded passed by Evaluation Modification as directed by the AVO. This test checks that a generic subprogram's formal parameter names (i.e. both generic and subprogram formal parameter names) must be distinct; the duplicated names within the generic declarations are marked as errors, whereas their recurrences in the subprogram bodies are marked as "optional" errors--except for the case at line 122, which is marked as an error. This implementation does not additionally flag the errors in the bodies and thus the expected error at line 122 is not flagged. The AVO ruled that the implementation's behavior was acceptable and that the test need not be split (such a split would simply duplicate the case in B83E01A at line 15).

CA2009A, CA2009C..D (2 tests), CA2009F and BC3009C were graded inapplicable by Evaluation Modification as directed by the AVO. These tests instantiate generic units before those units' bodies are compiled; this implementation creates dependences as allowed by AI-00408 & AI-00506 such that the compilation of the generic unit bodies makes the instantiating units obsolete, and the objectives of these tests cannot be met.

BC3204C and BC3205D were graded passed by Processing Modification as directed by the AVO. These tests check that instantiations of generic units with unconstrained types as generic actual parameters are illegal if the generic bodies contain uses of the types that require a constraint. However, the generic bodies are compiled after the units that contain the instantiations, and this implementation creates a dependence of the instantiating units on the generic units as allowed by AI-00408 & AI-00506 such that the compilation of the generic bodies makes the instantiating units obsolete--no errors are detected. The processing of these tests was modified by compiling the seperate files in the following order (to allow re-compilation of obsolete units), and all intended errors were then detected by the compiler:

BC3204C: C0, C1, C2, C3M, C4, C5, C6, C3M

BC3205D: D0, D2, D1M

BC3204D and BC3205C were graded passed by Test Modification as directed by the AVO. These tests are similar to BC3204C and BC3205D above, except that all compilation units are contained in a single compilation. For these two tests, a copy of the main procedure (which later units make obsolete) was appended to the tests; all expected errors were then detected.

CD2A53A was graded inapplicable by Evaluation Modification as directed by the AVO. The test contains a specification of a power-of-ten value as small for a fixed-point type. The AVO ruled that, under ACVC 1.11, support of decimal smalls may be omitted.

AD9001B and AD9004A were graded passed by Processing Modification as directed by the AVO. These tests check that various subprograms may be interfaced to external routines (and hence have no Ada bodies). This implementation requires that a file specification exists for the foreign subprogram bodies. The following command was issued to the Librarian to inform it that the foreign bodies will be supplied at link time (as the bodies are not actually needed by the program, this command alone is sufficient:

```
ALC30 interface/system AD9004A
```

CE2103A, CE2103B and CE3107A were graded inapplicable by Evaluation Modification as directed by the AVO. The tests abort with an unhandled exception when USE_ERROR is raised on the attempt to create an external file. This is acceptable behavior because this implementation does not support external files (cf. AI-00332).

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For a point of contact for technical information about this Ada implementation system, see:

Mr Ron Duursma
Director of Ada Products
Tartan Inc.
300, Oxford Drive,
Monroeville, PA 15146,
USA.
Tel. (412) 856-3600

For a point of contact for sales information about this Ada implementation system, see:

Mr Bill Geese
Director of Sales
Tartan Inc.
300, Oxford Drive,
Monroeville, PA 15146,
USA.
Tel. (412) 856-3600

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming

Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

a) Total Number of Applicable Tests	3452	
b) Total Number of Withdrawn Tests	83	
c) Processed Inapplicable Tests	86	
d) Non-Processed I/O Tests	264	
e) Non-Processed Floating-Point Precision Tests	285	
f) Total Number of Inapplicable Tests	635	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

The above number of I/O tests were not processed because this implementation does not support a file system. The above number of floating-point tests were not processed because they used floating-point precision exceeding that supported by the implementation. When this compiler was tested, the tests 21isted in Section 2.1 had been withdrawn because of test errors.
22

3.3 TEST EXECUTION

Version 1.11 of the ACVC comprises 4170 tests. When this compiler was tested, the tests listed in Section 2.1 had been withdrawn because of test errors. The AVF determined that 635 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 285 executable tests that use floating-point precision exceeding that supported by the implementation and 264 executable tests that use file operations not supported by the implementation. In addition, the modified tests mentioned in Section 2.3 were also processed.

A Magnetic Tape Reel containing the customized test suite (see Section 1.3) was taken on-site by the validation team for processing. The contents of the tape were loaded directly onto the host computer.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

The tests were compiled and linked on the host computer system, as appropriate. The executable images were transferred to the target computer system by the communications link, an RS232 Interface, and run. The results were captured on the host computer system.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the

default options. The options invoked explicitly for validation testing during this test were:

options used for compiling:

/replace	forces the compiler to accept an attempt to compile a unit imported from another library which is normally prohibited.
/nosave_source	suppresses the creation of a registered copy of the source code in the library directory for use by the REMAKE and MAKE subcommands.
/list=always	forces a listing to be produced, default is to only produce a listing when an error occurs.

No explicit Linker options were used.

Test output, compiler and linker listings, and job logs were captured on a Magnetic Tape Reel and archived at the AVF. The listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for $MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
$MAX_IN_LEN	240
$ACC_SIZE	32
$ALIGNMENT	1
$COUNT_LAST	2147483646
$DEFAULT_MEM_SIZE	16777216
$DEFAULT_STOR_UNIT	32
$DEFAULT_SYS_NAME	TI320C30
$DELTA_DOC	2#1.0#E-31
$ENTRY_ADDRESS	SYSTEM.ADDRESS'(16#809803#)
$ENTRY_ADDRESS1	SYSTEM.ADDRESS'(16#809804#)
$ENTRY_ADDRESS2	SYSTEM.ADDRESS'(16#809805#)
$FIELD_LAST	20
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_TYPE
$FLOAT_NAME	NO_SUCH_TYPE
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	100_000.0
$GREATER_THAN_DURATION_BASE_LAST	131_073.0
$GREATER_THAN_FLOAT_BASE_LAST	3.50282E+38
$GREATER_THAN_FLOAT_SAFE_LARGE	1.0E+38

```
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE
                        1.0E+38

$HIGH_PRIORITY          100

$ILLEGAL_EXTERNAL_FILE_NAME1
                        ILLEGAL_EXTERNAL_FILE_NAME1

$ILLEGAL_EXTERNAL_FILE_NAME2
                        ILLEGAL_EXTERNAL_FILE_NAME2

$INAPPROPRIATE_LINE_LENGTH
                        -1

$INAPPROPRIATE_PAGE_LENGTH
                        -1

$INCLUDE_PRAGMA1        "PRAGMA INCLUDE ("A28006D1.TST")"

$INCLUDE_PRAGMA2        "PRAGMA INCLUDE ("B28006F1.TST")"

$INTEGER_FIRST          -2147483648

$INTEGER_LAST           2147483647

$INTEGER_LAST_PLUS_1    2147483648

$INTERFACE_LANGUAGE     Ti_C

$LESS_THAN_DURATION     -100_000.0

$LESS_THAN_DURATION_BASE_FIRST
                        -131_073.0

$LINE_TERMINATOR        ' '

$LOW_PRIORITY           10

$MACHINE_CODE_STATEMENT
                        Two_Opnds'(LDI,(Imm,5),(Reg,R0));

$MACHINE_CODE_TYPE      Instruction_Mnemonic

$MANTISSA_DOC           31

$MAX_DIGITS             9

$MAX_INT                2147483647

$MAX_INT_PLUS_1         2147483648

$MIN_INT                -2147483648
```

```
$NAME                   NO_SUCH_TYPE_AVAILABLE

$NAME_LIST              TI320C30

$NAME_SPECIFICATION1    DUA2:[ACVC11.C30.TESTBED]X2120A.;1

$NAME_SPECIFICATION2    DUA2:[ACVC11.C30.TESTBED]X2120B.;1

$NAME_SPECIFICATION3    DUA2:[ACVC11.C30.TESTBED]X3119A.;1

$NEG_BASED_INT          16#FFFFFFFE#

$NEW_MEM_SIZE           16777216

$NEW_STOR_UNIT          32

$NEW_SYS_NAME           TI320C30

$PAGE_TERMINATOR        ' '

$RECORD_DEFINITION      record Operation: Instruction_Mnemonic;
                        Operand_1: Operand; Operand_2: Operand;
                        end record;

$RECORD_NAME            Two_Opnds

$TASK_SIZE              32

$TASK_STORAGE_SIZE      4096

$TICK                   0.00006103515625

$VARIABLE_ADDRESS       SYSTEM.ADDRESS'(16#809800#)

$VARIABLE_ADDRESS1      SYSTEM.ADDRESS'(16#809801#)

$VARIABLE_ADDRESS2      SYSTEM.ADDRESS'(16#809802#)

$YOUR_PRAGMA            NO_SUCH_PRAGMA
```

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

Compilation switches for Tartan Ada VMS C30.

/NOCROSS_REFERENCE [default]
Controls whether the compiler generates a cross-reference table of linknames for the compilation unit. The table will be placed in the file unit-name.XRF (See Section 3.6).

/CALLSHORTRANGE
Generates 16-bit PC-relative conditional call instructions. By using this option, the user asserts that the program space for the final program will be small enough for all calls to use the 16-bit PC-relative conditional call instruction. If the assertion is in fact incorrect, erroneous code could result.

/ERROR_LIMIT=n
Stop compilation and produce a listing after n errors are encountered, where n is in the range 0..255. The default value for n is 255. The /ERROR_LIMIT qualifier cannot be negated.

/FIXUP[=option]
When package MACHINE_CODE is used, controls whether the compiler attempts to alter operand address modes when those address modes are used incorrectly. The available options are:

QUIET
The compiler attempts to generate extra instructions to fix incorrect address modes in the array aggregates operand field.

WARN
The compiler attempts to generate extra instructions to fix incorrect address modes. A warning message is issued if such a ''fixup'' is required.

NONE
The compiler does not attempt to fix any machine code insertion that has incorrect address modes. An error message is issued for any machine code insertion that is incorrect.

When no form of this qualifier is supplied in the command line, the default condition is /FIXUP=QUIET. For more information on machine code insertions, refer to section 5.10 of this manual.

/LIBRARY=library-name
Specifies the library into which the file is to be compiled. The compiler reads any

ADALIB.INI files in the default directory.

/LIST[=option]
/NOLIST

Controls whether a listing file is produced. If produced, the file has the source file name and a .LIS extension. The available options are:

ALWAYS	Always produce a listing file
NEVER	Never produce a listing file, equivalent to /NOLIST
ERROR	Produce a listing file only if a compilation error or warning occurs

When no form of this qualifier is supplied in the command line, the default condition is /LIST=ERROR. When the LIST qualifier is supplied without an option, the default option is ALWAYS.

/MACHINE_CODE
/NOMACHINE_CODE [default]

Controls whether the assembly code files produced by the compiler are retained in the user's directory after compilation is complete. This qualifier is useful if the user wishes to inspect the compiler output for code correctness and quality. The default is /NOMACHINE which deletes these machine language files.

/NOENUMIMAGE

Causes the compiler to omit data segments with the text of enumeration literals. This text is normally produced for exported enumeration types in order to support the text attributes ('IMAGE, 'VALUE and 'WIDTH). You should use /NOENUMIMAGE only when you can guarantee that no unit that will import the enumeration type will use any of its text attributes. However, if you are compiling a unit with an enumeration type that is not visible to other compilation units, this qualifier is not needed. The compiler can recognize when the text attributes are not used and will not generate the supporting strings.The /NOENUMIMAGE qualifier cannot be negated.

/NOHUGELOOPS

Inform the compiler that no loops will iterate more than 2**23 times. This includes non-user specified loops, such as those generated by the compiler to operate on large objects. If the assertion is in fact, incorrect,

erroneous code could result.

/NODELAYEDBRANCHES — Do not generate delayed branch instructions.

/OPT=n — Controls the level of optimization performed by the compiler, requested by n. The /OPT qualifier cannot be negated. The optimization levels available are:

- n = 0 — Minimum - Performs context determination, constant folding, algebraic manipulation, and short circuit analysis. Inlines are not expanded.
- n = 1 — Low - Performs level 0 optimizations plus common subexpression elimination and equivalence propagation within basic blocks. It also optimizes evaluation order. Inlines are not expanded.
- n = 2 — Best tradeoff for space/time - the default level. Performs level 1 optimizations plus flow analysis which is used for common subexpression elimination and equivalence propagation across basic blocks. It also performs invariant expression hoisting, dead code elimination, and assignment killing. Level 2 also performs lifetime analysis to improve register allocation. It also performs inline expansion of subprogram calls indicated by Pragma INLINE, if possible.
- n = 3 — Time - Performs level 2 optimizations plus inline expansion of subprogram calls which the optimizer decides are profitable to expand (from an execution time perspective). Other optimizations which improve execution time at a cost to image size are performed only at this level.
- n = 4 — Space - Performs those

optimizations which usually produce the smallest code, often at the expense of speed. This optimization level may not always produce the smallest code, however, another level may produce smaller code under certain conditions.

/PHASES
/NOPHASES [default] Controls whether the compiler announces each phase of processing as it occurs. These phases indicate progress of the compilation. If there is an error in compilation, the error message will direct users to a specific location as opposed to the more general /PHASES.

/SUPPRESS[=(option, ...)]

Suppresses the specific checks identified by the options supplied. The parentheses may be omitted if only one option is supplied. The /SUPPRESS qualifier has the same effect as a global pragma SUPPRESS applied to the source file. If the source program also contains a pragma SUPPRESS, then a given check is suppressed if either the pragma or the qualifier specifies it; that is, the effect of a pragma SUPPRESS cannot be negated with the command line qualifier. The /SUPPRESS qualifier cannot be negated.

The available options are:

ALL	Suppress all checks. This is the default if the qualifier is supplied with no option.
ACCESS_CHECK	As specified in the Ada LRM, Section 11.7.
CONSTRAINT_CHECK	Equivalent of (ACCESS_CHECK, INDEX_CHECK, DISCRIMINANT_CHECK, LENGTH_CHECK, RANGE_CHECK).
DISCRIMINANT_CHECK	As specified in

allocator as a result of a new operation.

/WAITSTATES=[1..7] — Specify the number of wait states for the memory in which the branch code will be generated. It affects the code generated for the delayed branches and for loops. The default value is 2 wait states.

/WARNINGS [default]
/NOWARNINGS — Controls whether the warning messages generated by the compiler are displayed to the user at the terminal and in a listing file, if produced. While supressing warning messages also halts display of informational messages, it does not suppress Error, Fatal_Error.

	the Ada LRM, Section 11.7.
DIVISION_CHECK	Will suppress compile-time checks for division by zero, but the hardware does not permit efficient runtime checks, so none are done.
ELABORATION_CHECK	As specified in the Ada LRM, Section 11.7.
INDEX_CHECK	As specified in the Ada LRM, Section 11.7.
LENGTH_CHECK	As specified in the Ada LRM, Section 11.7.
NONE	No checks are suppressed. This is the default if the qualifier is not supplied on the command line.
OVERFLOW_CHECK	Will suppress compile-time checks for overflow, but the hardware does not permit efficient runtime checks, so none are done.
RANGE_CHECK	As specified in the Ada LRM, Section 11.7.
STORAGE_CHECK	As specified in the Ada LRM, Section 11.7. Suppresses only stack checks in generated code, not the checks made by the

LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to linker documentation and not to this report.

Linker switches for VMS hosted Tartan Ada compilers.

COMMAND QUALIFIERS

This section describes the command qualifiers available to a user who directly invokes the linker. The qualifier names can be abbreviated to unique prefixes; the first letter is sufficent for all current qualifier names. The qualifier names are not case sensitive.

/CONTROL=file	**The specified file contains linker control commands. Only one such file may be specified, but it can include other files using the CONTROL command. Every invocation of the linker must specify a control file.**
/OUTPUT=file	**The specified file is the name of the first output object file. The module name for this file will be null. Only one output file may be specified in this manner. Additional output files may be specified in the linker control file.**
/ALLOCATIONS	**Produce a link map showing the section allocations.**
/UNUSEDSECTIONS	**Produce a link map showing the unused sections.**
/SYMBOLS	**Produce a link map showing global and external symbols.**
/RESOLVEMODULES	**This causes the linker to not perform unused section elimination. Specifying this option will generally make your program larger, since unreferenced data within object files will not be eliminated. Refer to Sections RESOLVE_CMD and USE_PROCESSING for information on the way that unused section elimination works.**
/MAP	**Produce a link map containing all information except the unused section listings.**

Note that several listing options are permitted. This is because link maps for real systems can become rather large, and writing them consumes a significant fraction of the total link time. Options specifying the contents of the link map can be combined, in which case the resulting map will contain all the information specified by any of the switches. The name of the file containing the link map is specified by the LIST command in the linker control file. If your control file does not specify a name and you request a listing, the listing will be written to the default output stream.

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is

    ..........

    type INTEGER is range -2147483648 .. 2147483647;

    type FLOAT   is digits 6 range
         -2#1.00000000000000000000000#e+128 ..
          2#0.11111111111111111111111#e+128;

    type LONG_FLOAT   is digits 9 range
          -2#1.00000000000000000000000000000000#e+128 ..
           2#1.11111111111111111111111111111111#e+128;

    type DURATION is delta 0.0001 range -86400.0 .. 86400.0;


        ..........

    end STANDARD;
```

Chapter 4
Appendix F to MIL-STD-1815A

This chapter contains the required Appendix F to the LRM which is *Military Standard, Ada Programming Language*, ANSI/MIL-STD-1815A (American National Standards Institute, Inc., February 17, 1983).

4.1. *PRAGMAS*

4.1.1. Predefined Pragmas

This section summarizes the effects of and restrictions on predefined pragmas.

- Access collections are not subject to automatic storage reclamation so pragma CONTROLLED has no effect. Space deallocated by means of UNCHECKED_DEALLOCATION will be reused by the allocation of new objects.
- Pragma ELABORATE is supported.
- Pragma INLINE is supported.
- Pragma INTERFACE is supported. The Language_Name TI_C is used to make calls to subprograms (written in the Texas Instruments *C* language) from Tartan Ada. Any other Language_Name will be accepted, but ignored, and the default will be used.
- Pragma LIST is supported but has the intended effect only if the command qualifier LIST=ALWAYS was supplied for compilation, and the listing generated was not due to the presence of errors and/or warnings.
- Pragma MEMORY_SIZE is accepted but no value other than that specified in Package SYSTEM (Section 4.3) is allowed.
- Pragma OPTIMIZE is supported except when at the outer level (that is, in a package specification or body).
- Pragma PACK is supported.
- Pragma PAGE is supported but has the intended effect only if the command qualifier LIST=ALWAYS was supplied for compilation, and the listing generated was not due to the presence of errors and/or warnings.
- Pragma PRIORITY is supported.
- Pragma STORAGE_UNIT is accepted but no value other than that specified in Package SYSTEM (Section 4.3) is allowed.
- Pragma SHARED is not supported. No warning is issued if it is supplied.
- Pragma SUPPRESS is supported.
- Pragma SYSTEM_NAME is accepted but no value other than that specified in Package SYSTEM (Section 4.3) is allowed.

4.1.2.1. Pragma LINKAGE_NAME

The pragma LINKAGE_NAME associates an Ada entity with a string that is meaningful externally; e.g., to a linkage editor. It takes the form

pragma LINKAGE_NAME (*Ada-simple-name, string-constant*)

The *Ada-simple-name* must be the name of an Ada entity declared in a package specification. This entity must be one that has a runtime representation; e.g., a subprogram, exception or object. It may not be a named number or string constant. The pragma must appear after the declaration of the entity in the same package specification.

The effect of the pragma is to cause the *string-constant* to be used in the generated assembly code as an external name for the associated Ada entity. It is the responsibility of the user to guarantee that this string constant is meaningful to the linkage editor and that no illegal linkname clashes arise.

This pragma has no effect when applied to a library subprogram or to a *renames* declaration; in the latter case, no warning message is given.

When determining the maximum allowable length for the external linkage name, keep in mind that the compiler will generate names for elaboration flags simply by appending the suffix #GOTO. Therefore, the external linkage name has 5 fewer significant characters than the lower limit of other tools that need to process the name (e.g., 40 in the case of the Tartan Linker).

4.1.2.2. Pragma FOREIGN_BODY

In addition to Pragma INTERFACE, Tartan Ada supplies Pragma FOREIGN_BODY as a way to access subprograms in other languages.

Unlike Pragma INTERFACE, Pragma FOREIGN_BODY allows access to objects and exceptions (in addition to subprograms) to and from other languages.

Some restrictions on Pragma FOREIGN_BODY that are not applicable to Pragma INTERFACE are:

- Pragma FOREIGN_BODY must appear in a non-generic library package.
- All objects, exceptions and subprograms in such a package must be supplied by a foreign object module.
- Types may not be declared in such a package.

Use of the pragma FOREIGN_BODY dictates that all subprograms, exceptions and objects in the package are provided by means of a foreign object module. In order to successfully link a program including a foreign body, the object module for that body must be provided to the library using the ALBC30 FOREIGN_BODY command described in Section LIB-FOREIGN. The pragma is of the form:

pragma FOREIGN_BODY (*Language_name* [, *elaboration_routine_name*])

The parameter *Language_name* is a string intended to allow the compiler to identify the calling convention used by the foreign module (but this functionality is not yet in operation). Currently, the programmer must ensure that the calling convention and data representation of the foreign body procedures are compatible with those used by the Tartan Ada compiler. Subprograms called by tasks should be reentrant.

The optional *elaboration_routine_name* string argument is a linkage name identifying a routine to initialize the package. The routine specified as the *elaboration_routine_name*, which will be called for the elaboration of this package body, must be a global routine in the object module provided by the user.

A specification that uses this pragma may contain only subprogram declarations, object decarations that use an unconstrained type mark, and number declarations. Pragmas may also appear in the package. The type mark for an object cannot be a task type, and the object declaration must not have an initial value expression. The pragma must be given prior to any declarations within the package specification. If the pragma is not located before the first declaration, or any restriction on the declarations is violated, the pragma is ignored and a warning is generated.

The foreign body is entirely responsible for initializing objects declared in a package utilizing pragma FOREIGN_BODY. In particular, the user should be aware that the implicit initializations described in LRM 3.2.1 are not done by the compiler. (These implicit initializations are associated with objects of access types, certain record types and composite types containing components of the preceding kinds of types.)

Pragma LINKAGE_NAME should be used for all declarations in the package, including any declarations in a nested package specification to be sure that there are no conflicting link names. If pragma LINKAGE_NAME is not used, the cross-reference qualifier, /CROSS_REFERENCE, (see Section 3.2) should be used when invoking the compiler and the resulting cross-reference table of linknames inspected to identify the linknames assigned by the compiler and determine that there are no conflicting linknames (see also Section 3.5). In the following example, we want to call a function plmn which computes polynomials and is written in C.

```
package MATH_FUNCTIONS is
  pragma FOREIGN_BODY ("C");
  function POLYNOMIAL  (X:INTEGER) return INTEGER;
        --Ada spec matching the C routine
  pragma LINKAGE_NAME (POLYNOMIAL, "plmn");
        --Force compiler to use name "plmn" when referring to this
  -- function
end MATH_FUNCTIONS;

with MATH_FUNCTIONS; use MATH_FUNCTIONS;
procedure MAIN is
  X:INTEGER := POLYNOMIAL(10);
        -- Will generate a call to "plmn"
  begin ...
end MAIN;
```

To compile, link and run the above program, you do the following steps:

1. Compile MATH_FUNCTIONS
2. Compile MAIN
3. Obtain an object module (e.g. math.TOF) containing the compiled code for plmn.
4. Issue the command

```
ALBC30 FOREIGN_BODY math_functions MATH.TOF
```

5. Issue the command

```
ALBC30 LINK MAIN
```

Without Step 4, an attempt to link will produce an error message informing you of a missing package body for MATH_FUNCTIONS.

Using an Ada body from another Ada program library. The user may compile a body written in Ada for a specification into the library, regardless of the language specified in the pragma contained in the specification. This capability is useful for rapid prototyping, where an Ada package may serve to provide a simulated response for the functionality that a foreign body may eventually produce. It also allows the user to replace a foreign body with an Ada body without recompiling the specification.

The user can either compile an Ada body into the library, or use the command ALBC30 FOREIGN_BODY (see Section LIB-FOREIGN) to use an Ada body from another library. The Ada body from another library must have been compiled under an identical specification. The pragma LINKAGE_NAME must have been applied to all entities declared in the specification. The only way to specify the linkname for the elaboration routine of an Ada body is with the pragma FOREIGN_BODY.

4.2. *IMPLEMENTATION-DEPENDENT ATTRIBUTES*

No implementation-dependent attributes are currently supported.

4.3. *SPECIFICATION OF THE PACKAGE* SYSTEM

The parameter values specified for the Texas Instruments 320C30 processor family target in package SYSTEM [LRM 13.7.1 and Annex C] are:

```
package SYSTEM is
   type ADDRESS is new INTEGER;
   type NAME is (TI320C30);
   SYSTEM_NAME   : constant NAME := TI320C30;
   STORAGE_UNIT  : constant := 32;
   MEMORY_SIZE   : constant := 16_777_216;
   MAX_INT       : constant := 2_147_483_647;
   MIN_INT       : constant := -MAX_INT - 1;
   MAX_DIGITS    : constant := 9;

   MAX_MANTISSA  : constant := 31;
   FINE_DELTA    : constant := 2#1.0#e-31;
   TICK          : constant := 0.00006103515625  -- 2**(-14)
   subtype PRIORITY is INTEGER range 10 .. 100;
   DEFAULT_PRIORITY : constant PRIORITY := PRIORITY'FIRST;
   RUNTIME_ERROR    : exception;
end SYSTEM;
```

4.4. *RESTRICTIONS ON REPRESENTATION CLAUSES*

The following sections explain the basic restrictions for representation specifications followed by additional restrictions applying to specific kinds of clauses.

4.4.1. Basic Restriction

The basic restriction on representation specifications [LRM 13.1] is that they may be given only for types declared in terms of a type definition, excluding a `generic_type_definition` (LRM 12.1) and a `private_type_definition` (LRM 7.4). Any representation clause in violation of these rules is not obeyed by the compiler; an error message is issued.

Further restrictions are explained in the following sections. Any representation clauses violating those restrictions cause compilation to stop and a diagnostic message to be issued.

4.4.2. Length Clauses

Length clauses [LRM 13.2] are, in general, supported. For details, refer to the following sections.

4.4.2.1. Size Specifications for Types

The rules and restrictions for size specifications applied to types of various classes are described below.

The following principle rules apply:

1. The size is specified in bits and must be given by a static expression.
2. The specified size is taken as a mandate to store objects of the type in the given size wherever feasible. No attempt is made to store values of the type in a smaller size, even if possible. The following rules apply with regard to feasibility:

   - An object that is not a component of a composite object is allocated with a size and alignment that is referable on the target machine; that is, no attempt is made to create objects of non-referable size on the stack. If such stack compression is desired, it can be achieved by the user by combining multiple stack variables in a composite object; for example

```
type My_Enum is (A,B);
for My_enum'size use 1;
V,W: My_enum; -- will occupy two storage
              -- units on the stack
              -- (if allocated at all)
type rec is record
   V,W: My_enum;
end record;
pragma Pack(rec);
O: rec;       -- will occupy one storage unit
```

- A formal parameter of the type is sized according to calling conventions rather than size specifications of the type. Appropriate size conversions upon parameter passing take place automatically and are transparent to the user.

- Adjacent bits to an object that is a component of a composite object, but whose size is non-referable, may be affected by assignments to the object, unless these bits are occupied by other components of the composite object; that is, whenever possible, a component of non-referable size is made referable.

In all cases, the compiler generates correct code for all operations on objects of the type, even if they are stored with differing representational sizes in different contexts.

Note: A size specification cannot be used to force a certain size in value operations of the type; for example

```
type my_int is range 0..65535;
for my_int'size use 16; -- o.k.
A,B: my_int;
   ...A + B... -- this operation will generally be
               -- executed on 32-bit values
```

3. A size specification for a type specifies the size for objects of this type and of all its subtypes. For components of composite types, whose subtype would allow a shorter representation of the component, no attempt is made to take advantage of such shorter representations. In contrast, for types without a length clause, such components may be represented in a lesser number of bits than the number of bits required to represent all values of the type. Thus, in the example

```
type MY_INT is range 0..2**15-1;
for MY_INT'SIZE use 16; -- (1)
subtype SMALL_MY_INT is MY_INT range 0..255;
type R is record
            ...
           X: SMALL_MY_INT;
            ...
         end record;
```

the component R.X will occupy 16 bits. In the absence of the length clause at (1), R.X may be represented in 8 bits.

Size specifications for access types must coincide with the default size chosen by the complier for the type.

Size specifications are not supported for floating-point types or task types.

No useful effect can be achieved by using size specifications for these types.

4.4.2.2. Size Specification for Scalar Types

The specified size must accommodate all possible values of the type including the value 0 (even if 0 is not in the range of the values of the type). For numeric types with negative values the number of bits must account for the sign bit. No skewing of the representation is attempted. Thus

```
type my_int is range 100..101;
```

requires at least 7 bits, although it has only two values, while

```
type my_int is range -101..-100;
```

requires 8 bits to account for the sign bit.

A size specification for a real type does not affect the accuracy of operations on the type. Such influence should be exerted via the `accuracy_definition` of the type (LRM 3.5.7, 3.5.9).

A size specification for a scalar type may not specify a size larger than the largest operation size supported by the target architecture for the respective class of values of the type.

4.4.2.3. Size Specification for Array Types

A size specification for an array type must be large enough to accommodate all components of the array under the densest packing strategy. Any alignment constraints on the component type (see Section 4.4.7) must be met.

The size of the component type cannot be influenced by a length clause for an array. Within the limits of representing all possible values of the component subtype (but not necessarily of its type), the representation of components may, however, be reduced to the minimum number of bits, unless the component type carries a size specification.

If there is a size specification for the component type, but not for the array type, the component size is rounded up to a referable size, unless pragma PACK is given. This applies even to boolean types or other types that require only a single bit for the representation of all values.

4.4.2.4. Size Specification for Record Types

A size specification for a record type does not influence the default type mapping of a record type. The size must be at least as large as the number of bits determined by type mapping. Influence over packing of components can be exerted by means of (partial) record representation clauses or by Pragma PACK.

Neither the size of component types, nor the representation of component subtypes can be influenced by a length clause for a record.

The only implementation-dependent components allocated by Tartan Ada in records contain dope information for arrays whose bounds depend on discriminants of the record or contain relative offsets of components within a record layout for record components of dynamic size. These implementation-dependent components cannot be named or sized by the user.

A size specification cannot be applied to a record type with components of dynamically determined size.

Note: Size specifications for records can be used only to widen the representation accomplished by padding at the beginning or end of the record. Any narrowing of the representation over default type mapping must be accomplished by representation clauses or pragma PACK.

4.4.2.5. Specification of Collection Sizes

The specification of a collection size causes the collection to be allocated with the specified size. It is expressed in storage units and need not be static; refer to package SYSTEM for the meaning of storage units.

Any attempt to allocate more objects than the collection can hold causes a STORAGE_ERROR exception to be raised. Dynamically sized records or arrays may carry hidden administrative storage requirements that must be accounted for as part of the collection size. Moreover, alignment constraints on the type of the allocated objects may make it impossible to use all memory locations of the allocated collection. No matter what the requested object size, the allocator must allocate a minimum of 2 words per object. This lower limit is necessary for administrative overhead in the allocator. For example, a request of 5 words results in an allocation of 5 words; a request of 1 word results in an allocation of 2 words.

In the absence of a specification of a collection size, the collection is extended automatically if more objects are allocated than possible in the collection originally allocated with the compiler-established default size. In this case, STORAGE_ERROR is raised only when the available target memory is exhausted. If a collection size of zero is specified, no access collection is allocated.

4.4.2.6. Specification of Task Activation Size

The specification of a task activation size causes the task activation to be allocated with the specified size. It is expressed in storage units; refer to package SYSTEM for the meaning of storage units.

Any attempt to exceed the activation size during execution causes a STORAGE_ERROR exception to be raised. Unlike collections, there is no extension of task activations.

4.4.2.7. Specification of 'SMALL

Only powers of 2 are allowed for 'SMALL.

The length of the representation may be affected by this specification. If a size specification is also given for the type, the size specification takes precedence; the specification of 'SMALL must then be accommodatable within the specified size.

4.4.3. Enumeration Representation Clauses

For enumeration representation clauses [LRM 13.3], the following restrictions apply:

- The internal codes specified for the literals of the enumeration type may be any integer value between INTEGER'FIRST and INTEGER'LAST. It is strongly advised to not provide a representation clause that merely duplicates the default mapping of enumeration types, which assigns consecutive numbers in ascending order starting with 0, since unnecessary runtime cost is incurred by such duplication. It should be noted that the use of attributes on enumeration types with user-specified encodings is costly at run time.
- Array types, whose index type is an enumeration type with non-contiguous value encodings, consist of a contiguous sequence of components. Indexing into the array involves a runtime translation of the index value into the corresponding position value of the enumeration type.

4.4.4. Record Representation Clauses

The alignment clause of record representation clauses [LRM 13.4] is observed.

Static objects may be aligned at powers of 2. The specified alignment becomes the minimum alignment of the record type, unless the minimum alignment of the record forced by the component allocation and the minimum alignment requirements of the components is already more stringent than the specified alignment.

The component clauses of record representation clauses are allowed only for components and discriminants of statically determinable size. Not all components need to be present. Component clauses for components of variant parts are allowed only if the size of the record type is statically determinable for every variant.

The size specified for each component must be sufficient to allocate all possible values of the component subtype (but not necessarily the component type). The location specified must be compatible with any alignment constraints of the component type; an alignment constraint on a component type may cause an implicit alignment constraint on the record type itself.

If some, but not all, discriminants and components of a record type are described by a component clause, then the discriminants and components without component clauses are allocated after those with component clauses; no attempt is made to utilize gaps left by the user-provided allocation.

4.4.5. Address clauses

Address clauses [LRM 13.5] are supported with the following restrictions:

- When applied to an object, an address clause becomes a linker directive to allocate the object at the given address. For any object not declared immediately within a top-level library package, the address clause is meaningless. Address clauses applied to local packages are not supported by Tartan Ada. Address clauses applied to library packages are prohibited by the syntax; therefore, an address clause can be applied to a package only if it is a body stub.

- Address clauses applied to subprograms and tasks are implemented according to the LRM rules. When applied to an entry, the specified value identifies an interrupt in a manner customary for the target. Immediately after a task is created, a runtime call is made for each of its entries having an address clause, establishing the proper binding between the entry and the interrupt.
- A specified address must be an Ada static expression.

4.4.6. Pragma PACK

Pragma PACK [LRM 13.1] is supported. For details, refer to the following sections.

4.4.6.1. Pragma PACK *for Arrays*

If pragma PACK is applied to an array, the densest possible representation is chosen. For details of packing, refer to the explanation of size specifications for arrays (Section 4.4.2.3).

If, in addition, a length clause is applied to

1. The array type, the pragma has no effect, since such a length clause already uniquely determines the array packing method.
2. The component type, the array is packed densely, observing the component's length clause. Note that the component length clause may have the effect of preventing the compiler from packing as densely as would be the default if pragma PACK is applied where there was no length clause given for the component type.

4.4.6.2. The Predefined Type String

Package STANDARD applies Pragma PACK to the type `string`. However, because type character is determined to be 32 bits on the C30, this results in one character per word.

4.4.6.3. Pragma PACK *for Records*

If pragma PACK is applied to a record, the densest possible representation is chosen that is compatible with the sizes and alignment constraints of the individual component types. Pragma PACK has an effect only if the sizes of some component types are specified explicitly by size specifications and are of non-referable nature. In the absence of pragma PACK, such components generally consume a referable amount of space.

It should be noted that the default type mapping for records maps components of boolean or other types that require only a single bit to a single bit in the record layout, if there are multiple such components in a record. Otherwise, it allocates a referable amount of storage to the component.

If pragma PACK is applied to a record for which a record representation clause has been given detailing the allocation of some but not all components, the pragma PACK affects only the components whose allocation has not been detailed. Moreover, the strategy of not utilizing gaps between explicitly allocated components still applies.

4.4.7. Minimal Alignment for Types

Certain alignment properties of values of certain types are enforced by the type mapping rules. Any representation specification that cannot be satisfied within these constraints is not obeyed by the compiler and is appropriately diagnosed.

Alignment constraints are caused by properties of the target architecture, most notably by the capability to extract non-aligned component values from composite values in a reasonably efficient manner. Typically, restrictions exist that make extraction of values that cross certain address boundaries very expensive, especially in contexts involving array indexing. Permitting data layouts that require such complicated extractions may impact code quality on a broader scale than merely in the local context of such extractions.

Instead of describing the precise algorithm of establishing the minimal alignment of types, we provide the general rule that is being enforced by the alignment rules:

- No object of scalar type including components or subcomponents of a composite type, may span a target-dependent address boundary that would mandate an extraction of the object's value to be performed by two or more extractions.

4.5. *IMPLEMENTATION-GENERATED COMPONENTS IN RECORDS*

The only implementation-dependent components allocated by Tartan Ada in records contain dope information for arrays whose bounds depend on discriminants of the record. These components cannot be named by the user.

4.6. *INTERPRETATION OF EXPRESSIONS APPEARING IN ADDRESS CLAUSES*

Section 13.5.1 of the Ada Language Reference Manual describes a syntax for associating interrupts with task entries. Tartan Ada implements the address clause

```
for TOENTRY use at intID;
```

by associating the interrupt specified by `intID` with the `toentry` entry of the task containing this address clause. The interpretation of `intID` is both machine and compiler dependent.

4.7. *RESTRICTIONS ON UNCHECKED CONVERSIONS*

Tartan supports `UNCHECKED_CONVERSION` with a restriction that requires the sizes of both source and target types to be known at compile time. The sizes need not be the same. If the value in the source is wider than that in the target, the source value will be truncated. If narrower, it will be zero-extended. Calls on instantiations of `UNCHECKED_CONVERSION` are made inline automatically.

4.8. *IMPLEMENTATION-DEPENDENT ASPECTS OF INPUT-OUTPUT PACKAGES*

Tartan Ada supplies the predefined input/output packages `DIRECT_IO`, `SEQUENTIAL_IO`, `TEXT_IO`, and `LOW_LEVEL_IO` as required by LRM Chapter 14. However, since the 320C30 chip is used in embedded applications lacking both standard I/O devices and file systems, the functionality of `DIRECT_IO`, `SEQUENTIAL_IO`, and `TEXT_IO` is limited.

`DIRECT_IO` and `SEQUENTIAL_IO` raise `USE_ERROR` if a file open or file access is attempted. `TEXT_IO` is supported to `CURRENT_OUTPUT` and from `CURRENT_INPUT`. A routine that takes explicit file names raises `USE_ERROR`.

4.9. *OTHER IMPLEMENTATION CHARACTERISTICS*

The following information is supplied in addition to that required by Appendix F to MIL-STD-1815A.

4.9.1. Definition of a Main Program

Any Ada library subprogram unit may be designated the main program for purposes of linking (using the ALBC30 `LINK` command) provided that the subprogram has no parameters.

Tasks initiated in imported library units follow the same rules for termination as other tasks [described in LRM 9.4 (6-10)]. Specifically, these tasks are not terminated simply because the main program has terminated. Terminate alternatives in selective wait statements in library tasks are therefore strongly recommended.

4.9.2. Implementation of Generic Units

All instantiations of generic units, except the predefined generic UNCHECKED_CONVERSION and UNCHECKED_DEALLOCATION subprograms, are implemented by code duplications. No attempt at sharing code by multiple instantiations is made in this release of Tartan Ada.

Tartan Ada enforces the restriction that the body of a generic unit must be compiled before the unit can be instantiated. It does not impose the restriction that the specification and body of a generic unit must be provided as part of the same compilation. A recompilation of the body of a generic unit will casue any units that instantiated this generic unit to become obsolete.

4.9.3. Attributes of Type Duration

The type DURATION is defined with the following characteristics:

Attribute	Value
DURATION'DELTA	0.0001 sec
DURATION'SMALL	$6.103516E^{-5}$ sec
DURATION'FIRST	-86400.0 sec
DURATION'LAST	86400.0 sec

4.9.4. Values of Integer Attributes

Tartan Ada supports the predefined integer type INTEGER. The range bounds of the predefined type INTEGER are:

Attribute	Value
INTEGER'FIRST	-2**31
INTEGER'LAST	2**31-1

The range bounds for subtypes declared in package TEXT_IO are:

Attribute	Value
COUNT'FIRST	0
COUNT'LAST	INTEGER'LAST - 1
POSITIVE_COUNT'FIRST	1
POSITIVE_COUNT'LAST	INTEGER'LAST - 1
FIELD'FIRST	0
FIELD'LAST	20

The range bounds for subtypes declared in packages DIRECT_IO are:

Attribute	Value
COUNT'FIRST	0
COUNT'LAST	INTEGER'LAST
POSITIVE_COUNT'FIRST	1
POSITIVE_COUNT'LAST	COUNT'LAST

4.9.5. Values of Floating-Point Attributes

Tartan Ada supports the predefined floating-point types FLOAT and LONG_FLOAT.

Attribute	Value for FLOAT
DIGITS	6
MANTISSA	23
EMAX	92
EPSILON	16#0.1000_00#E-4 (approximately 9.53674E-07)
SMALL	16#0.8000_00#E-21 (approximately 2.58494E-26)
LARGE	16#0.FFFF_F8#E+21 (approximately 1.93428E+25)
SAFE_EMAX	126
SAFE_SMALL	16#0.2000_00#E-31 (approximately 5.87747E-39)
SAFE_LARGE	16#0.3FFF_FE#E+32 (approximately 8.50706E+37)
FIRST	-16#0.1000_00#E+33 (approximately -3.40282E+38)
LAST	16#0.FFFF_FF#E+32 (approximately 3.40282E+38)
MACHINE_RADIX	2
MACHINE_MANTISSA	24
MACHINE_EMAX	128
MACHINE_EMIN	-126
MACHINE_ROUNDS	FALSE
MACHINE_OVERFLOWS	TRUE

Attribute	Value for LONG_FLOAT
DIGITS	9
MANTISSA	31
EMAX	124
EPSILON	16#0.4000_0000_0#E-7 (approximately 9.31322575E-10)
SMALL	16#0.8000_0000_0#E-31 (approximately 2.35098870E-38)
LARGE	16#0.FFFF_FFFE_0#E+31 (approximately 2.12676479E+37)
SAFE_EMAX	126
SAFE_SMALL	16#0.2000_0000_0#E-31 (approximately 5.87747175E-39)
SAFE_LARGE	16#0.3FFF_FFFF_8#E+32 (approximately 8.50705917E+37)
FIRST	-16#0.1000_0000_0#E+33 (approximately -3.40282367E+38)
LAST	16#0.FFFF_FFFF_0#E+32 (approximately 3.40282367E+38)
MACHINE_RADIX	2
MACHINE_MANTISSA	32
MACHINE_EMAX	128
MACHINE_EMIN	-126
MACHINE_ROUNDS	FALSE
MACHINE_OVERFLOWS	TRUE

4.10. *SUPPORT FOR PACKAGE* MACHINE_CODE

Package MACHINE_CODE provides the programmer with an interface through which to request the generation of any instruction that is available on the C30. The implementation of package MACHINE_CODE is similar to that described in Section 13.8 of the Ada LRM, with several added features. Please refer to appendix A for the Package MACHINE_CODE specification.

4.10.1. Basic Information

As required by LRM, Section 13.8, a routine which contains machine code inserts may not have any other kind of statement, and may not contain an exception handler. The only allowed declarative item is a use clause. Comments and pragmas are allowed as usual.

4.10.2. Instructions

A machine code insert has the form TYPE_MARK'RECORD_AGGREGATE, where the type must be one of the records defined in package MACHINE_CODE. Package MACHINE_CODE defines seven types of records. Each has an opcode and zero to 6 operands. These records are adequate for the expression of all instructions provided by the C30.

4.10.3. Operands and Address Modes

An operand consists of a record aggregate which holds all the information to specify it to the compiler. All operands have an address mode and one or more other pieces of information. The operands correspond exactly to the operands of the instruction being generated.

Each operand in a machine code insert must have an *Address_Mode_Name*. The address modes provided in package MACHINE_CODE provide access to all address modes supported by the C30.

In addition, package MACHINE_CODE supplies the address modes Symbolic_Address and Symbolic_Value which allow the user to refer to Ada objects by specifying Object'ADDRESS as the value for the operand. Any Ada object which has the 'ADDRESS attribute may be used in a symbolic operand. Symbolic_Address should be used when the operand is a true address (that is, a branch target for example). Symbolic_Value should be used when the operand is actually a value (that is, one of the source operands of an ADDI instruction).

When an Ada object is used as a *source* operand in an instruction (that is, one from which a value is read), the compiler will generate code which fetches the *value* of the Ada object. When an Ada object is used as the destination operand of an instruction, the compiler will generate code which uses the *address* of the Ada object as the destination of the instruction.

4.10.4. Examples

The implementation of package MACHINE_CODE makes it possible to specify both simple machine code inserts such as

```
Two_Opnds'(LDI, (Imm, 5), (Reg, R0))
```

and more complex inserts such as

```
Three_Opnds'(ADDI3,
        (Imm, 10),
        (Symbolic_Value, Array_Var(X, Y, 27)'ADDRESS),
        (Symbolic_Address, Parameter_1'ADDRESS))
```

In the first example, the compiler will emit the instruction LDI 5, R0. In the second example, the compiler will first emit an instruction to load the immediate value 10 into a register, next emit whatever instructions are needed to form the address of Array_Var(X, Y, 27) and then emit the ADDI3 instruction. If Parameter_1 is not found in a register, the compiler will put the result of the addition in a temporary register and then store it to Parameter_1'ADDRESS. Note that the destination operand of the ADDI3 instruction is given as a Symbolic_Address. This holds true for all destination operands. The various error checks specified in the LRM will be performed on all compiler-generated code unless they are suppressed by the programmer (either through pragma SUPPRESS, or through command qualifiers).

4.10.5. Incorrect Operands

Under some circumstances, the compiler attempts to correct incorrect operands. Three modes of operation are supplied for package MACHINE_CODE: /FIXUP=NONE, /FIXUP=WARN, and /FIXUP=QUIET. These modes of operation determine whether corrections are attempted and how much information about the necessary corrections is provided to the user. /FIXUP=QUIET is the default.

In /FIXUP=NONE mode, the specification of incorrect operands for an instruction is considered to be a fatal error. In this mode, the compiler will not generate any extra instructions to help you to make a machine code insertion. Note that it is still legal to use 'ADDRESS constructs as long as the object which is used meets the requirements of the instruction.

In /FIXUP=QUIET mode, if you specify incorrect operands for an instruction, the compiler will do its best to correct the machine code to provide the desired effect. For example, although it is illegal to use a memory address as the destination of an ADDI instruction, the compiler will accept it and try to generate correct code. In this case, the compiler will load the value found at the memory address indicated into a register, use this register in the ADDI instruction, and then store from that register back to the desired memory location.

```
Two_Opnds'(ADDI, (Imm, 10), (ARI, AR1))

will produce a code sequence like

LDI     *AR1, R0
ADDI    10, R0
STI     R0, *AR1
```

The next example illustrates the correction required when the displacement is out of range for the first operand of an ADDI3 instruction. The displacement is first loaded into one of the index registers.

```
Three_Opnds'(ADDI3, (IPDA, AR3, 2), (Reg, R0), (Reg, R1))

will produce a code sequence like

LDI     2, IR0
ADDI3   AR3(IR0), R0, R1
```

In /FIXUP=WARN mode, the compiler will also do its best to correct any incorrect operands for an instruction. However, a warning message is issued stating that the machine code insert required additional machine instructions to make its operands legal.

4.10.6. Assumptions Made in Correcting Operands

When compiling in /FIXUP=QUIET or /FIXUP=WARN modes, the compiler attempts to emit additional code to move "the right bits" from an incorrect operand to a place which is a legal operand for the requested instruction. The compiler makes certain basic assumptions when performing these corrections. This section explains the assumptions the compiler makes and their implications for the generated code. Note that if you want a correction which is different from that performed by the compiler, you must make explicit machine code insertions to perform it.

For source operands:

- Symbolic_Address means that the *address* specified by the 'ADDRESS expression is used as the source bits. When the Ada object specified by the 'ADDRESS instruction is bound to a register, this will cause a compile-time error message because it is not possible to "take the address" of a register.
- Symbolic_Value means that the *value* found at the address specified by the 'ADDRESS expression will be used as the source bits. An Ada object which is bound to a register is correct here, because the contents of a register can be expressed on the C30.
- PcRel indicates that the *address* of the label will be used as the source bits.
- Any other non-register means that the *value* found at the address specified by the operand will be used as the source bits.

For destination operands:

- Symbolic_Address means that the desired destination for the operation is the *address* specified by the 'ADDRESS expression. An Ada object which is bound to a register is correct here; a register is a legal destination on the C30.
- Symbolic_Value means that the desired destination for the operations is found by fetching 32 bits from the address specified by the 'ADDRESS expression, and storing the result to the address represented by the fetched bits. This is equivalent to applying one extra indirection to the address used in the Symbolic_Address case.
- All other operands are interpreted as directly specifying the destination for the operation.

4.10.7. Register Usage

Since the compiler may need to allocate registers as temporary storage in machine code routines, there are some restrictions placed on your register usage. The compiler will automatically free all the registers which would be volatile across a call for your use (that is, R0..R3, AR0..AR2, IR0, IR1, RS, RC, RE, BK, and DP). If you reference any other register, the compiler will reserve it for your use until the end of the machine code routine. The compiler will *not* save the register automatically if this routine is inline expanded. This means that the first reference to a register which is not volatile across calls should be an instruction which saves its value in a safe place. The value of the register should be restored at the end of the machine code routine. This rule will help ensure correct operation of your machine code insert even if it is inline explaned in another routine.

However, the compiler will save the register automatically in the prolog code for the routine and restore it in the epilog code for the routine if the routine is *not* inline expanded.

As a result of freeing all volatile registers for the user, any parameters which were passed in registers will be moved to either a non volatile register or to memory. References to `PARAMETER'ADDRESS` in a machine code insert will then produce code that uses this register or memory location. This means that there is a possiblity of invalidating the value of some `'ADDRESS` expression if the non volatile register which it is bound to is used as a destination in some later machine code insert. In this case, any subsequent references to the `'ADDRESS` expression will cause the compiler to issue a warning message.

The compiler may need several registers to generate code for operand fixups in machine code inserts. If you use all the registers, corrections will not be possible. In general, when more registers are available to the compiler it is able to generate better code.

4.10.8. Data Directives

Two special instructions are included in package `Machine_Code` to allow the user to place data into the code stream. These two instructions are `DATA32` and `DATA64`. Each of these instructions can have 1 to 6 operands.

`DATA32` is used to place 32-bit data into the code stream. The value of an integer or 32-bit float, and the address of a label are the legal operands (i.e. operands whose address mode is either `Imm`, `FloatImm`, or `Symbolic_Address` of an Ada label).

```
        << L1 >>
        Three_Opnds'(DATA32,  (Symbolic_Address,  L2'Address),
                              (Symbolic_Address,  L3'Address),
                              (Symbolic_Address,  L4'Address));
        << L2 >>
        << L3 >>
        << L4 >>

        will produce a code sequence like

      L1:     .word L2
              .word L3
              .word L4
```

`DATA64` is used to place a 64-bit data into the code stream. The only legal operand is a floating literal (i.e operand whose address mode is `FloatImm`).

4.10.9. Inline Expansion

Routines which contain machine code inserts may be inline expanded into the bodies of other routines. This may happen under programmer control through the use of pragma `INLINE`, or at optimization levels 2 and 3 when the compiler selects that optimization as an appropriate action for the given situation. The compiler will treat the machine code insert as a call; volatile registers will be saved and restored around it, etc.

4.10.10. Unsafe Assumptions

There are a variety of assumptions which should *not* be made when writing machine code inserts. Violation of these assumptions may result in the generation of code which does not assemble or which may not function correctly.

- The compiler *will not* generate call site code for you if you emit a call instruction. You must save and restore any volatile registers which currently have values in them, etc. If the routine you call has `out` parameters, a large function return result, or an unconstrained result, it is your responsibility to emit the necessary instructions to deal with these constructs as the compiler expects. In other words, when you emit a call, you must follow the linkage conventions of the routine you are calling. For further details on call site code, see Sections 6.4, 6.5 and 6.6.

- Do not assume that the 'ADDRESS on Symbolic_Address or Symbolic_Value operands means that you are getting an ADDRESS to operate on. The Address- or Value-ness of an operand is determined by your choice of Symbolic_Address or Symbolic_Value. This means that to add the *contents* of X to AR0, you should write

```
Two_Opnds'(ADDI, (Symbolic_Value, X'ADDRESS),
                 (Reg, AR0))
```

but to add the *address* of X to AR0, you should write

```
Two_Opnds'(ADDI, (Symbolic_Address, X'ADDRESS),
                 (Reg, AR0));
```

4.10.11. Limitations

The current implementation of the compiler is unable to fully support automatic correction of certain kinds of operands. In particular, the compiler assumes that the size of a data object is the same as the number of bits which is operated on by the instruction chosen in the machine code insert. This means that the insert:

```
Two_Opnds'(ADDF, (Symbolic_Value, Long_Float_Variable'ADDRESS),
                 (Reg, R0))
```

will not generate correct code when Long_Float_Variable is bound to memory. The compiler will assume that Long_Float_Variable is 32 bits, when in fact it is stored in 64 bits of memory. If, on the other hand, Long_Float_Variable was bound to an extended-precision register, the insertion will function properly, as no correction is needed.

Note that the use of X'ADDRESS in a machine code insert *does not* guarantee that X will be bound to memory. This is a result of the use of 'ADDRESS to provide a "typeless" method for naming Ada objects in machine code inserts. For example, it is legal to say (Symbolc_Value, X'ADDRESS) in an insert even when X is found in a register.

4.10.12. Example

```
with machine_code; use machine_code;
procedure mach_example is

        type ary_type is array(1..4) of integer;

        a: ary_type := (1,2,3,4);
        b: integer;

        procedure casestatement(a: in integer; b: in out integer) is
        begin
        -- implements case a is
        --             when 1 => b := 0;
        --             when 2 => b := b + 1;
        --             when 3 => b := b * b;
        --             when others => null
        --             end case;
           Three_Opnds'(SUBI3, (Imm, 1), (Symbolic_Value, a'Address), (Reg, IR0));
           Two_Opnds'(LDI, (Symbolic_Address, L1'Address), (Reg, Ar0));
           Two_Opnds'(LDI, (IPrIA, Ar0, IR0), (Reg, Ar1));
           One_Opnds'(case_jump, (Reg, Ar1));
           << L1 >>
           Three_Opnds'(DATA32, (Symbolic_Address, L2'Address),
                                (Symbolic_Address, L3'Address),
                                (Symbolic_Address, L4'Address));
           << L2 >>
           Two_Opnds'(LDI, (Imm, 0), (Symbolic_Address, b'Address));
           One_Opnds'(BU, (PcRel, L5'Address));
           << L3 >>
           Two_Opnds'(ADDI, (Imm, 1), (Symbolic_Value, b'Address));
           One_Opnds'(BU, (PcRel, L5'Address));
           << L4 >>
```

```
            Two_Opnds'(MPYI, (Symbolic_Value, b'Address), (Symbolic_Value, b'Address));
            << L5 >>
            Zero_Opnds'(NOP);  -- since label can't be last statement in procedure
         end casestatement;

         pragma inline(casestatement);

begin
   if a(1) >= 0 then
     casestatement(a(3), b);  -- will be inline expanded
   end if;

end mach_example;
```

Assembly code output:

```
         .global mach_example
; mach_example.tmp from mtests/manualexample.ada
; Ada Sun/C30 Version V11.631293001 Copyright 1989, Tartan Laboratories
; 1986
         .global xxmchxmple008

         .text

xxmchxmple008:  PUSH    AR3
         LDIU    SP,AR3
         PUSH    AR3
         ADDI    4,SP
         PUSH    R6
         PUSH    R7
         PUSH    AR7
         LDIU    @DEF1,AR0
         STI     AR0,*+AR3(1)

         LDIU    @DEF2,AR0              ; line 7
         LDIU    AR3,AR1
         ADDI    2,AR1
         LDIU    *AR0++(1),R1
         RPTS    2
         LDI     *AR0++(1),R1
         || STI  R1,*AR1++(1)
         STI     R1,*AR1
         LDI     *+AR3(2),R0             ; line 43
         BLT     L22
         LDIU    *+AR3(4),AR7            ; line 44
         LDFU    R6,R7
         LDIU    R6,R7
         LDIU    1,R1            ; line 18
         LDIU    IR0,AR0
         SUBI3   R1,AR7,R2
         STI     R2,*AR0
         LDI     @DEF3,R0               ; line 19
         STI     R0,*AR0
         LDI     *+AR0(IR0),R2           ; line 20
         STI     R2,*AR1
         BU      AR1             ; line 21
L14:
         .word   L15

         .word   L16

         .word   L17

L15:     LDI     0,R7            ; line 27
         BU      L18             ; line 28
```

```
L16:    ADDI    1,R7            ; line 30
        BU      L18             ; line 31
L17:    MPYI    R7,R7           ; line 33
L18:    NOP                     ; line 35
        LDIU    R7,R6           ; line 44
L22:

        LDIU    *+AR3(6),R6
        LDIU    *+AR3(7),R7
        LDIU    *+AR3(8),AR7
        LDIU    AR3,SP
        POP     AR3
        RETSU

; Total words of code in the above routine = 46

        .data
DEF3:   .word   L14
DEF1:   .word   L22


        .text

casestatement$00:       RETSU

; Total words of code in the above routine = 1

        .data


        .text
        .data
DEF2:   .word   DEF4
DEF4:   .word   1
        .word   2
        .word   3
        .word   4

; Total words of code = 47
; Total words of data = 7

        .end
```

4.11. *DELAYED BRANCHES*

A feature of the C30 architecture is the inclusion of delayed branching. Because of the processor pipelining, normal branch instructions require four cycles to execute. During that time the pipeline is emptied and no other useful instructions may be executed. However, a second set of branch instructions is provided that allow three more instructions to be executed after initiation of the branch and before actual transfer of control. It is very important to use delayed branches whenever possible in order to achieve maximum processor throughput.

4.11.1. Generating Delayed Branches

A special machine-dependent optimization phase attempts to generate delayed branches by seeking to identify instructions that can be scheduled within the three-instruction branch delay. An instruction may be scheduled during the branch delay if it is:

1. An instruction that currently precedes the branch in the basic block and produces no result or side effect that could be used by any instruction preceding its potential location within the branch delay.
2. An instruction that currently follows a conditional branch, providing it has no side effects if the branch *is* taken and produces no result or side effect that could be mis-used by any instruction between its potential branch delay location and its current location.

3. A replication of an instruction that is currently at the address of the branch destination. If the branch is conditional, this instruction must have no side effects if the branch is *not* taken. The branch target address must be changed to point to the next address if such an instruction is discovered.

Instructions which are themselves branches may *not* be scheduled within the branch delay.

As an example, in the following code fragment, as presented to the delay branch optimization, can the BEQ be usefully transformed into the delayed version, BEQD?

```
No.     Instruction

10      LDI   0,R0              ; R0 := 0;
11      ADDI  1,R1              ; R1 := R1 + 1;
12      CMPI  R1,R2             ; compare R1 to R2
13      BEQ   L1                ; branch if equal to L1
14      LDI   *-AR1(1),R3       ; R3 := some memory value
    || LDI   *AR7++(IR0),R4     ; and in parallel also load R4
15      ADDI  R7,R5             ; R5 := R5 + R7;

        ...

21 L1:  LDI   33,R3             ; R3 := 33;
22      LDI   R4,R5             ; R4 := R5;
```

Searching for instructions in the first class above, the delayed branch optimizer discovers that instruction 10 can be moved down to the branch delay because its results are not used by instructions 11 or 12. Instruction 12 is not moveable since its result is used by the conditional branch. Likewise, instruction 11's result is used by instruction 12 and so it cannot be moved.

Applying the rules for the second type of delayed branch candidate, the delayed branch optimizer discovers that instruction 14 (a two operation parallel instruction) cannot be moved up into the branch delay since it loads R4 and the contents of R4 are read by instruction 22 if the branch is taken. Instruction 15 can be moved up into the branch delay because it produces no result that can affect instruction 14 and its results are voided by instruction 22 if the branch is taken.

While searching for the members of the third class, it is detected that instruction 21 can be replicated in the branch delay and the branch retargeted, because its result is voided by instruction 14 if the branch is not taken. Instruction 22 cannot be moved since it will leave R5 in the incorrect state for instruction 15.

After the transformations are made, the code is:

```
No.     Instruction

11      ADDI  1,R1              ; R1 := R1 + 1;
12      CMPI  R1,R2             ; compare R1 to R2
13      BEQD  L2                ; delay branch if equal L2
10      LDI   0,R0              ; R0 := 0;
15      ADDI  R7,R5             ; R5 := R5 + R7;
21      LDI   33,R3             ; R3 := 33;
                                ; branch takes effect here
14      LDI   *-AR1(1),R3       ; R3 := some memory value
    || LDI   *AR7++(IR0),R4     ; and in parallel also load R4

        ...

21 L1:  LDI   33,R3             ; R3 := 33;
22 L2:  LDI   R4,R5             ; R5 := R4;
```

Note that the code is now very obscure. Maintaining assembly code of this nature will indeed be difficult, and here we see a true advantage to the automated approach offered by the Ada compiler.

Many modern processor architectures contain delayed branch instructions. However, because the C30 allows for three instructions during the branch delay instead of the usual one, this feature is no longer transparent to the casual user of any C30 compiler. For example, the compiler may discover that only a single useful instruction can be executed during the branch delay and must "fill" the other two slots with no-op's. While this strategy may speed up the algorithm by one cycle, it also increases code size by two words. The tradeoff with two useful instructions and one no-op is a possible speed-up of two cycles for a code increase of one word. The Ada compiler allows the user to specify a global speed vs. space tradeoff strategy. This assertion combined with loop depth and other static program measurements will cause the compiler to bias against marginal delayed branch opportunities that expand the code. The compiler's stategy is discussed in detail below.

4.11.2. Delayed Branch Strategy

Certain conditions determine whether the delayed branch optimizer will actually transform a standard branch into its delayed branch equivalent. These conditions include whether or not the code is in fast memory, the optimization level used to compile the source code, how many useful instructions have been found to fill the delay slots, and where these instructions came from (i.e., above the branch, below the branch, or from the label). The compiler will transform a branch instruction only if the resulting code is guaranteed to be at least as fast as the original code.

The delayed branch optimizer makes decisions based on instruction fetch time. A program compiled with the switch `/WAITSTATES=0` tells the compiler that the code will be in fast memory. The compiler can also detect that an insruction will reside in the cache by checking if it is in a loop that completely fits in the cache. If the three delay slots that follow a delayed branch will not reside in fast memory or the cache, then three useful instructions must be found to fill the delay slots. No no-op instruction will be inserted to fill a delay slot because the resulting code could end up being slower than the code with the standard branch.

The compiler will not transform any standard branches into delayed branches at optimization levels 0 or 1. At optimization level 4 (Space), a standard branch is transformed into a delayed branch only if three instructions have been found to fill the delay slots. Optimization level 2 (Custom Mix) requires at least two useful instructions to fill the delay slots. Optimization level 3 (Speed) requires only one useful instruction. At these two levels, no-op instructions will be added to fill the remaining delay slots only if the code is in fast memory or the cache. If the code is not in fast memory or the cache, and three instructions have not been found, then the branch will not be transformed.

As stated above, the position of an instruction that can be moved to a delay slot with respect to the branch instruction itself has an effect on whether the branch will be transformed. It is always best to find useful instructions that would be executed regardless of the condition on the branch. Instructions that precede the branch fall into this category. Likewise, if the branch is an unconditional branch, instructions at the branch target address will also always be executed, so they also fall into this category. However, instructions that follow a conditional branch will only be executed if the branch is not taken, and instructions at the branch target address of a conditional branch will only be executed if the branch is taken. Therefore, it is not always beneficial to fill the delay slots with these kind of instructions. These instructions will fill a delay slot only when the resulting code is at least as fast as the orignal code regardless of whether the branch is taken or not. The exception to this is that instuctions below a conditional branch will be considered as always being executed when the delay branch optimizer can determine that the condition on the branch will not be satisfied a large percentage of the time.

4.12. *PACKAGE* `INTRINSICS`

The `Intrinsics` package is provided as a means for the programmer to access certain hardware capabilities of the 320C30 in an efficient manner.

The package declares generic functions which may be instantiated to create functions that have particularly efficient implementations. A call to such a function usually does not include a hardware subroutine call at all, but is implemented inline as a few 320C30 instructions. (Often a single instruction!)

4.12.1. Native Instructions

The following group of generic functions allows specific 320C30 instructions to be applied to Ada entities. The user must instantiate the generic function for the types that will be used as the operand(s) and result of the operation. These generic functions have been given the same name as the assembler's name for the corresponding instruction. In some cases this convention leads to a conflict with an Ada reserved word. This conflict is resolved by using the instruction name with an "`i`" appended to it.

For details of the operation applied by calling an instance of one of these generic functions see the *TMS320C30 User's Guide*. Examples of their use are given in figures 4-1 and 4-2. Refer to appendix B for the signatures of all intrinsics. The available operations are shown in the following table.

Name	Meaning
ANDi	bitwise logical-AND.
ANDN	bitwise logical AND-NOT.
ASH	arithmetic shift
FIX	floating point to integer conversion
FLOATi	integer to floating point conversion
LDE	load floating point exponent
LDM	load floating point mantissa
LSH	logical shift
MPYF	32-bit x 32-bit -> 40-bit floating multiply
MPYI	24-bit x 24-bit -> 32-bit integer multiply
NORM	floating point normalize
RND	round floating point
NOTi	bitwise logical complement
ORi	bitwise logical OR
ROL	rotate left
ROR	rotate right
SUBC	subtract integer conditionally
XORi	bitwise exclusive OR

```
with intrinsics; use intrinsics;
with int_io;
with text_io;
with flt_io;

procedure test(a : float; b : integer; c : integer) is

   function "*" is new MPYI(integer,integer,integer);
      -- All integer multiply operations in this subprogram
      -- will be done with the 24-bit native instruction.

   function SQRT is new SQRT_32;
begin
   text_io.put_line("The square root of a is ");
   flt_io.put(SQRT(a),8,6,0)
   text_io.new_line;

   text_io.put_line("b x c (24-bit x 24-bit -> 32 bit) is ");
   int_io.put(b * c, 2);
   text_io.new_line;
end test;
```

Figure 4-1: Example Use of Intrinsics

```
with intrinsics; use intrinsics;

function shift(
   ShiftMe : integer;
   ShiftCount : integer;
   Signed : boolean) return integer is
   -- Try writing this without these intrinsics!

   function    LogicalShift is new LSH(integer,integer);
   function ArithmeticShift is new ASH(integer,integer);
begin
   if Signed then return ArithmeticShift(ShiftMe, ShiftCount);
   else           return    LogicalShift(ShiftMe, ShiftCount);
   end if;
end test;
```

Figure 4-2: LSH and ASH Used To Define a General Purpose Shift Routine

4.12.2. Circular Addressing

The 320C30 circular addressing modes are made available through a set of generic functions that model the entire process with an iterator object and a set of subprograms to

- initialize the iterator,
- read an object specified by the current value of the iterator,
- advance the iterator to a new object, and
- release the iterator for later use.

These generics are documented using the same names and terms as in section 6.3 of the *TMS320C30 User's Guide*.

A fixed number of iterators are available for use in circular addressing. Iterators are named by the enumeration literals of the type `Circ_Iterator_Name_Type`. More than one iterator may be active at any given time.

There are certain very important rules to keep in mind when using circular iterators.

1. Initialization of an iterator must occur prior to usage ***both*** in execution order ***and*** textual order in the source code.
2. Similarly, release of an iterator must occur after last usage ***both*** in execution order ***and*** textual order in the source code.
3. The iterator is known only within the procedure in which it is defined.
4. Finally, because these functions are considered to be free of side effects, optimization may unfortunately remove a call to one if the result of the call does not seem necessary for the execution of the program. In particular, for each of the functions that returns a `Boolean` value, the call should be used as the condition in an `if` statement whose `then` part is an assignment to a non-local variable. The call will not be removed and the useless assignment will be correctly optimized away. See figure 4-3 for an example of how this is done.

The 320C30 hardware places certain requirements on the addresses used in circular addressing operations. The values passed to an instantiation of `Init_Circ_Iter`, the iterator initialization function, must obey these rules. This normally means that it is necessary to use an address clause to position the entity whose address is passed as the first parameter of `Init_Circ_Iter`.

```
-- FIR package demonstrating circular iterators

with System;
package Fir_Package is
 generic
    N : Integer; -- instantiate on a per-buffer-size basis
    type Vector is array (Positive range <>) of Float;
 procedure Fir (H: in Vector;
                AddressOfNextElement: in out System.Address;
                Y: in out Float);
 Dummy : Boolean;
end Fir_Package;

with Intrinsics; use Intrinsics;
with System;
package body Fir_Package is
 procedure Fir (H: in Vector;
                AddressOfNextElement: in out System.Address;
                Y: in out Float) is
    function Init        is new Init_Circ_Iter;
    function Release     is new Release_Circ_Iter;
    function ReadThenAdd is new Read_Then_Circ_Add(Float);
    function EBPlusIndex is new Circ_Iter_EB_Plus_Index;
    function CircAdd     is new Circ_Add;
 begin
    y := 0.0;

    if not Init(AddressOfNextElement, 1, N, Circ_Iter_1) then
       Dummy := False; -- no code generated
    end if;

    for Step in 0 .. N-1  loop
       Y := Y + ReadThenAdd(Circ_Iter_1) * H(Step);
    end loop;

    if not CircAdd(Circ_Iter_1) then Dummy := False; end if;

    AddressOfNextElement := EBPlusIndex(Circ_Iter_1);

    if not Release(Circ_Iter_1) then Dummy := False; end if;
 end Fir;
end Fir_Package;
```

Figure 4-3: FIR Package Demonstrating Circular Iterator Intrinsics

The functions that operate on circular iterators are:

Name	Meaning
`Init_Circ_Iter`	Allocate and initialize an iterator. From left to right, the parameters are:
	`EB_Plus_Start_Index` Usually `Array(Start_Index)'address`. Given *n* such that *n* is smallest value where `2**n` > BK, then `Array'address mod 2**n)` *must* = 0, which can be guaranteed only if `Array` is placed in memory using an Ada address clause.
	`Step` Usually `Array(0)'size/32`.
	`BK` Usually `Array'size`
	`Name` One of `Circ_Iterator_Name_Type`
`Release_Circ_Iter`	Returns boolean true. Releases the iterator resources to the compiler for other use.
`Read_Circ_Iter`	Returns the value pointed to by the current value of the iterator, plus some arbitrary integer offset.
`Read_Then_Circ_Add`	Returns the value pointed to by the current value of the iterator, then advances the iterator in accordance with the `Step` and `BK` specified in the initialization.
`Read_Then_Circ_Sub`	Returns the value pointed to by the current value of the iterator, then advances the iterator in accordance with the `Step` and `BK` specified in the initialization.
`Write_Circ_Iter`	Returns boolean true. Writes the location pointed to by the current value of the iterator, plus some arbitrary integer offset.
`Write_Then_Circ_Add`	Returns boolean true. Writes the location pointed to by the current value of the iterator, then advances the iterator in accordance with the Step and BK specified in the initialization.
`Write_Then_Circ_Sub`	Returns boolean true. Writes the location pointed to by the current value of the iterator, then advances the iterator in accordance with the Step and BK specified in the initialization.
`Circ_Add`	Returns boolean true. Advances the iterator in accordance with the `Step` and `BK` specified in the initialization.
`Circ_Sub`	Returns boolean true. Advances the iterator in accordance with the `Step` and `BK` specified in the initialization.
`Circ_Iter_EB_Plus_Index`	Extracts and returns the "`EB+Index`" part of the iterator.
`Circ_Iter_Step`	Extracts and returns the `Step` part of the iterator.
`Circ_Iter_BK`	Extracts and returns the `BK` part of the iterator.

Hints for Improved Object Code Quality:

- For improved code, initialize the "most important" iterators first in textual order in the source code.

- If all BK's of all active iterators are not proveably the same at compile-time, generated code will degrade considerably.
- Always release iterators when they are no longer needed.

4.12.3. Bit-Reversed Addressing

The 320C30 bit-reversed addressing modes are made available through a set of generic functions that model the entire process with an iterator object and a set of subprograms to

- initialize the iterator,
- read an object specified by the current value of the iterator,
- advance the iterator to a new object, and
- release the iterator for later use.

These generics are documented using the same names and terms as in section 6.4 of the *TMS320C30 User's Guide*.

A fixed number of iterators are available for use in bit-reversed addressing. Iterators are named by the enumeration literals of the type `Brev_Iterator_Name_Type`. More than one iterator may be active at any given time.

The same rules regarding the use of circular addressing iterators apply to bit-reversed addressing iterators.

The functions that operate on bit-reversed iterators are:

`Init_Brev_Iter` — Allocate and initialize an iterator. From left to right, the parameters are:

`Base_Addr_Plus_Start_Index` — Usually `Array(Start_Index, 0)'address`, assuming that the second dimension of the array holds the data points to be addressed between each change in the bit-reverse iterator. `Array'address` mod `Two_To_N` *must* = 0. This can be guaranteed only if `Array` is placed in memory using an Ada *address* clause.

`Two_To_N` — Usually `Array'size/2`. Must be a power of two.

`Name` — One of `Brev_Iterator_Name_Type`.

`Release_Brev_Iter` — Returns boolean **true**. Releases the iterator resources to the compiler for other use.

`Read_Brev_Iter` — Returns the value pointed to by the current value of the iterator, plus some arbitrary integer offset.

`Read_Then_Brev_Add` — Returns the value pointed to by the current value of the iterator, then advances the iterator according to the `Two_To_N` specified in the initialization.

`Write_Brev_Iter` — Returns boolean **true**. Writes the location pointed to by the current value of the iterator, plus some arbitrary integer offset.

`Write_Then_Brev_Add` — Returns boolean **true**. Writes the location pointed to by the current value of the iterator, then advances the iterator in accordance with the `Two_To_N` specified in the initialization.

`Brev_Add` — Returns boolean **true**. Advances the iterator in accordance with the `Two_To_N` specified in the initialization.

`Brev_Iter_BA_Plus_Index`	Extracts and returns the "`Base_Addres` + `Index`" part of the iterator.
`Brev_Iter_Two_To_N`	Extracts and returns the "`Two_To_N`" part of the iterator.

The example given for circular iterators (figure 4-3) is a good guide for the use of bit-reversed iterators as well.

4.12.4. Mathematical Functions

Access to a limited set of standard floating-point functions is provided by the following generic functions.

```
generic function SQRT_32   (X : Float) return Float;
generic function ALOG_32   (X : Float) return Float;
generic function ALOG10_32 (X : Float) return Float;
generic function EXP_32    (X : Float) return Float;
generic function TENTO_32  (X : Float) return Float;
generic function POWER_32  (X,Y:Float) return Float;

generic function SIN_32    (X : Float) return Float;
generic function COS_32    (X : Float) return Float;
generic function TAN_32    (X : Float) return Float;
generic function COT_32    (X : Float) return Float;
generic function ASIN_32   (X : Float) return Float;
generic function ACOS_32   (X : Float) return Float;
generic function ATAN_32   (X : Float) return Float;
generic function ATAN2_32  (X,Y:Float) return Float;
generic function ACOT_32   (X : Float) return Float;
generic function ACOT2_32  (X,Y:Float) return Float;
generic function SINH_32   (X : Float) return Float;
generic function COSH_32   (X : Float) return Float;
generic function TANH_32   (X : Float) return Float;
generic function COTH_32   (X : Float) return Float;
generic function ASINH_32  (X : Float) return Float;
generic function ACOSH_32  (X : Float) return Float;
generic function ATANH_32  (X : Float) return Float;
generic function ACOTH_32  (X : Float) return Float;
```

```
with intrinsics; use intrinsics;
with text_io;
with flt_io;

procedure test(a : float) is
   function "**" is new POWER_32;
   function ALOG10 is new ALOG10_32;
   b : float;
begin
   text_io.put_line("test: a, 10**a, alog10(10**a):");
   flt_io.put(a,8,6,0);
   b := 10**a;
   flt_io.put(b, 8,6,0);
   flt_io.put(alog10(b),8,6,0);
   text_io.new_line;
end test;
```

Figure 4-4: Using the Intrinsic Math Functions

Figure 4-4 shows an example of the use of POWER_32 instantiated as the "**" operator for floats. A call to an instance of any of these results in a call to an extremely fast-executing function to perform the computation. These are "shared-code" generics in the sense that there will be only one object-code version of each function created no matter how many instantiations are made.

The code generator contains built-in knowledge that these function calls are free from side effects and thus do not cause optimizations to be blocked. The code generator also knows exactly which of the volatile registers are used by each routine and will not save active values from registers that are not used by the routine being called.

Algorithms for the routines were adapted from *Software Manual for the Elementary Functions*; Cody and Waite, Prentice Hall 1980; and *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*; Milton Abramowitz and Irene A. Stegun, National Bureau of Standards (Applied Mathematics Series 55), Washington D.C., 1964 (reprinted 1970); and the *TMS320C30 User's Guide*. Some algorithms were developed internally.

All routines are accurate to single (32-bit) floating precision. An augmented set of Cody-Waite accuracy tests has been used to test them. Loss of precision was found to be limited to about 2 bits of the 24-bit mantissa for almost all of the functions. Test results are available from Tartan on request.

Every attempt has been made to avoid raising an exception for any input value. Reasonable values are returned under all conditions. It is assumed that most signal processing applications work in a "press on" mode.

In the table that follows, cycles were counted by hand and are based on 0-wait state memory for both program and data spaces. The min count is for non-trivial case(s). The range over which each count holds is usually documented. The "typical" count assumes uniform distribution of input values across stated range. When range is not declared, the typical count holds for range of all 32-bit floating numbers.

Routine	Min Cycles	Max Cycles	Typical Cycles
SQRT	37	37	37
ALOG	35	107 s2m1<|x|<s2	37 |x| <= 100
ALOG10	36	108 s2m1<|x|<s2	38 |x| <= 100
EXP	31	31	31
TENTO	35	35	35
POWER	119	124	122
SIN	29 |x| < ts	34	29 |x| <= ts
COS	28 |x| < tc	33	28 |x| <= tc
TAN	51 |x| < tt	56	53 |x| <= tt
COT	53	58	53
ASIN	56 |x| <= .5	74 |x| > .5	65
ACOS	57 |x| <= .5	75 |x| > .5	66
ATAN	33	63	48
ATAN2	75	105	90
ACOT	35	65	50
ACOT2	78	108	93
SINH	48 |x| <= 1.0	83 1<|x|<=te	66 |x| <= 2.0
COSH	38 te<|x|<te69	79 |x|<=te	79 |x| <= te
TANH	53 |x|<=.549	82	75 |x| <= 2.196
COTH	86	115	108 |x| >= 0.455
ASINH	19 |x|<=.5,else 97	169 -1 < x < -0.5	97 |x| <= 100
ACOSH	87	159 x < tac	87 x <= 100
ATANH	25 |x| <= .5	85 |x| > .5	55
ACOTH	54 |x| >= 2	85 |x| < 2	70

KEY:

```
s2m1 = SQRT(2) - 1.0
s2   = SQRT(2)
tc   = 2**24 * PI - PI/2 = 52707176.96 ~= 52707000
ts   = 2**24 * PI        = 52707178.53 ~= 52707000
tt   = (2**24 -.5) * PI/2= 26353588.48 ~= 26353000
te   = 88.71875
te69 = 88.71875 + 0.69316 = 89.41191
tac  = 3*SQRT(2) / 4 = 1.060660172
```